МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

*«Программирование многопоточных приложений. POSIX Threads*»

студента 2 курса, группы 22206

***Тропина Никиты Васильевича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

ассистент,

А.А.Ажбаков

Новосибирск 2023

# ЦЕЛЬ

Освоить разработку многопоточных программ с использованием POSIX Threads API. Познакомиться с задачей динамического распределения работы между процессами.

# ЗАДАНИЕ

Написать программу с использованием POSIX Threads API, симулирующую работу кластера.

Программа не должна зависеть от количества процессов.

Использование коммуникаций MPI между процессами.

Использование POSIX Threads для порождения нитей в рамках процесса.

В каждом процесса минимум 2 нити.

Вес задания считать заранее неизвестным для процессов.

# ОПИСАНИЕ РАБОТЫ

Задания из списка могут быть выполнены независимо друг от друга. Задания могут иметь различный вычислительный вес, т.е. требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Этот вес нельзя узнать, пока задание не выполнено. После того, как все задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах. Количество заданий существенно превосходит количество процессоров.

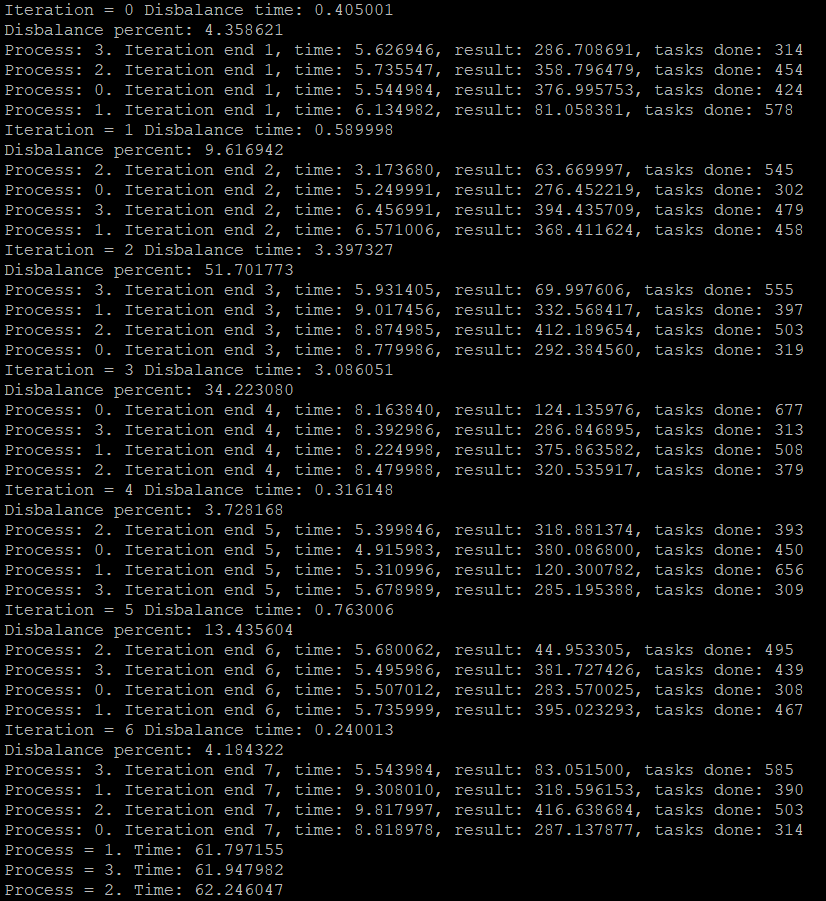
Программа не должна зависеть от числа компьютеров. Так как задания имеют различный вычислительный вес, а список обрабатывается итеративно, и требуется синхронизация перед каждой итерацией, то могут возникать ситуации, когда некоторые процессоры выполнили свою работу, а другие - еще нет. Если ничего не предпринять, первые будут простаивать в ожидании последних. Так возникает задача динамического распределения работы. Для ее решения на каждом процессоре заведем несколько потоков. Как минимум, потоков должно быть 2:

* поток, который обрабатывает задания и, когда задания закончились, обращается к другим компьютерам за добавкой к работе,
* поток, ожидающий запросов о работе от других компьютеров

Необходимо обеспечивать взаимное исключение потоков при добавлении заданий в список, удалении задач, выборке заданий для выполнения.

# ЗАКЛЮЧЕНИЕ

Вывод программы оказался следующим:



Процент дисбаланса оказался достаточно малым, значит программа корректно распределяет нагрузку между процессами.

# ПРИЛОЖЕНИЕ

#include <iostream>

#include <mpi.h>

#include <cmath>

#include <unistd.h>

#include <vector>

#include <pthread.h>

#include <cstdio>

struct Task {

int repeat\_num;

};

const int THREAD\_NUM = 2;

const int L = 10000;

const int ITERATIONS\_COUNT = 8;

pthread\_mutex\_t mutex;

pthread\_t threads[THREAD\_NUM];

std::vector<Task> tasks;

MPI\_Datatype TASK\_TYPE;

int rank;

int size;

int iter\_counter;

int task\_index;

int get\_new\_task(int proc\_rank);

double calc\_task(Task& task);

void generate\_tasks(int count\_tasks);

void\* calc\_thread(\_\_attribute\_\_((unused)) void\* args);

void\* data\_thread(\_\_attribute\_\_((unused)) void\* args);

int run();

int main(int argc, char\*\* argv) {

int provided;

MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, &provided);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

if (provided != MPI\_THREAD\_MULTIPLE) {

MPI\_Finalize();

return EXIT\_FAILURE;

}

// Создание нового типа для отправки структуры Task

int count = 1;

int block\_lengths[] = { 1 };

MPI\_Datatype types[] = { MPI\_INT };

MPI\_Aint displs = 0;

MPI\_Type\_create\_struct(count, block\_lengths, &displs, types, &TASK\_TYPE);

MPI\_Type\_commit(&TASK\_TYPE);

pthread\_mutex\_init(&mutex, nullptr);

double start = MPI\_Wtime();

if (run() != EXIT\_SUCCESS) {

pthread\_mutex\_destroy(&mutex);

MPI\_Type\_free(&TASK\_TYPE);

MPI\_Finalize();

return EXIT\_FAILURE;

}

double end = MPI\_Wtime();

printf("Process = %d. Time: %lf\n", rank, end - start);

pthread\_mutex\_destroy(&mutex);

MPI\_Type\_free(&TASK\_TYPE);

MPI\_Finalize();

return EXIT\_SUCCESS;

}

int run() {

pthread\_attr\_t attrs;

if (pthread\_attr\_init(&attrs) != 0)

{

perror("Error during init attrs");

return EXIT\_FAILURE;

}

if (pthread\_attr\_setdetachstate(&attrs, PTHREAD\_CREATE\_JOINABLE) != 0)

{

perror("Error setting attr joinable");

abort();

}

pthread\_create(&threads[0], &attrs, calc\_thread, nullptr);

pthread\_create(&threads[1], &attrs, data\_thread, nullptr);

pthread\_attr\_destroy(&attrs);

for (auto& thread : threads) {

if (pthread\_join(thread, nullptr) != 0) {

perror("Error joining thread");

return EXIT\_FAILURE;

}

}

return EXIT\_SUCCESS;

}

void\* calc\_thread(\_\_attribute\_\_((unused)) void\* args) {

int\* able\_get\_task\_flags = new int[size];

for (iter\_counter = 0; iter\_counter < ITERATIONS\_COUNT; iter\_counter++) {

task\_index = 0;

double result = 0;

int task\_count = 0;

int count = 100 \* size;

generate\_tasks(count);

double start = MPI\_Wtime();

for (int i = 0; i < size; i++) {

able\_get\_task\_flags[i] = 1;

}

able\_get\_task\_flags[rank] = 0;

while (true) {

while (task\_index < tasks.size()) {

pthread\_mutex\_lock(&mutex);

Task task\_to\_do = tasks[task\_index];

pthread\_mutex\_unlock(&mutex);

result += calc\_task(task\_to\_do);

task\_count++;

task\_index++;

}

int i = 0;

for (; i < size;) {

if (!able\_get\_task\_flags[i]) {

i++;

continue;

}

else {

task\_count++;

able\_get\_task\_flags[i] = get\_new\_task(i);

break;

}

}

if (i == size) {

break;

}

}

double end = MPI\_Wtime();

double time = end - start;

printf("Process: %d. Iteration end %d, time: %lf, "

"result: %lf, tasks done: %d\n", rank, iter\_counter, time, result, task\_count);

MPI\_Barrier(MPI\_COMM\_WORLD);

double max\_time = 0;

double min\_time = 0;

MPI\_Reduce(&time, &max\_time, 1, MPI\_DOUBLE, MPI\_MAX, 0,

MPI\_COMM\_WORLD);

MPI\_Reduce(&time, &min\_time, 1, MPI\_DOUBLE, MPI\_MIN, 0,

MPI\_COMM\_WORLD);

if (rank == 0) {

printf("Iteration = %d Disbalance time: %lf\nDisbalance percent: %lf\n",

iter\_counter, max\_time - min\_time, (max\_time - min\_time) / max\_time \* 100.0);

}

}

int request = 0;

MPI\_Send(&request, 1, MPI\_INT, rank, 1, MPI\_COMM\_WORLD);

delete[] able\_get\_task\_flags;

return nullptr;

}

void\* data\_thread(\_\_attribute\_\_((unused)) void\* args) {

while (iter\_counter < ITERATIONS\_COUNT) {

MPI\_Status status;

int res;

MPI\_Recv(&res, 1, MPI\_INT, MPI\_ANY\_SOURCE, 1, MPI\_COMM\_WORLD,

&status);

if (res == 0) break;

if (task\_index >= tasks.size()) {

int answer = 0;

MPI\_Send(&answer, 1, MPI\_INT, status.MPI\_SOURCE, 2,

MPI\_COMM\_WORLD);

}

else {

pthread\_mutex\_lock(&mutex);

Task task\_to\_send = tasks.back();

tasks.pop\_back();

pthread\_mutex\_unlock(&mutex);

int answer = 1;

MPI\_Send(&answer, 1, MPI\_INT, status.MPI\_SOURCE, 2, MPI\_COMM\_WORLD);

MPI\_Send(&task\_to\_send, 1, TASK\_TYPE, status.MPI\_SOURCE, 3, MPI\_COMM\_WORLD);

}

}

return nullptr;

}

int get\_new\_task(int proc\_rank) {

int request = 1;

MPI\_Send(&request, 1, MPI\_INT, proc\_rank, 1, MPI\_COMM\_WORLD);

MPI\_Recv(&request, 1, MPI\_INT, proc\_rank, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

if (request == 0) return 0;

Task task{};

MPI\_Recv(&task, 1, TASK\_TYPE, proc\_rank, 3, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

pthread\_mutex\_lock(&mutex);

tasks.push\_back(task);

pthread\_mutex\_unlock(&mutex);

return 1;

}

double calc\_task(Task& task) {

double res = 0;

for (int i = 0; i < task.repeat\_num; i++) {

res += sin(i);

}

return res;

}

void generate\_tasks(int count\_tasks)

{

pthread\_mutex\_lock(&mutex);

tasks.clear();

for (int i = 0; i < count\_tasks; i++) {

Task task{};

task.repeat\_num = std::abs(50 - i % 100) \* std::abs(rank - (iter\_counter % size)) \* L;

tasks.push\_back(task);

}

pthread\_mutex\_unlock(&mutex);

}